# Scans in an embedded hardware design language

By Yorick Sijsling

Supervised by Wouter Swierstra

With assistance of João Paulo Pizani Flor

Colloquium UU CS - May 1st 2015

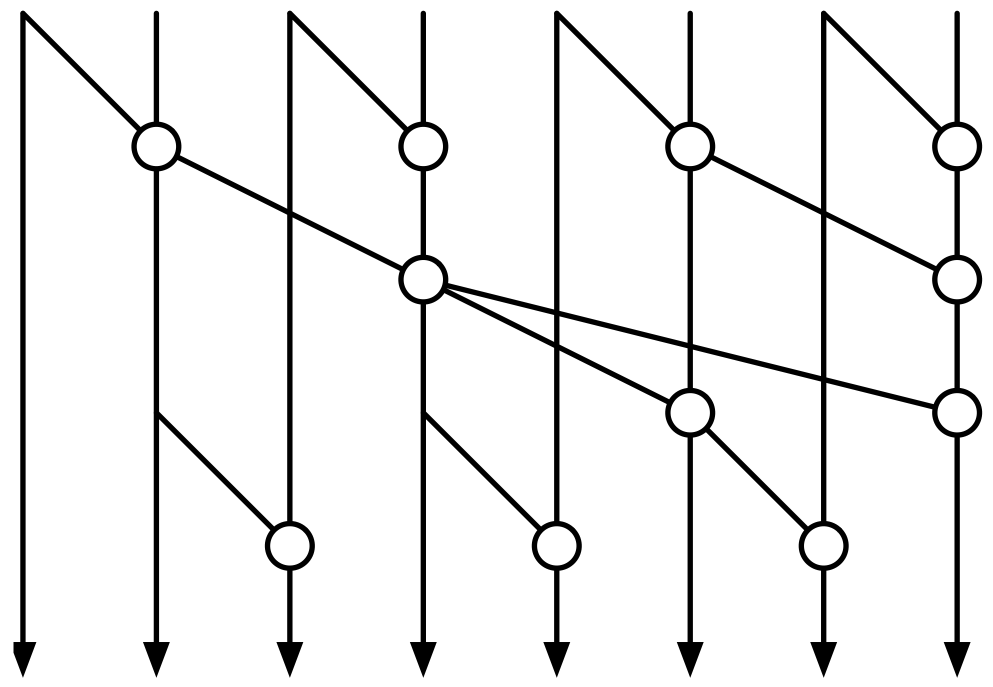
# Scans

- Also known as *Parallel prefix circuits*
- Takes an *associative* binary operator

$$\begin{tikzcd}
x_1, x_2, ..., x_n \arrow[d, "scan\ \oplus"] \\
x_1, (x_1 \oplus x_2), \cdots, (x_1 \oplus x_2 \oplus \cdots \oplus x_n)
\end{tikzcd}$$

# Scan circuits

$x_1, x_2, \dots, x_n$

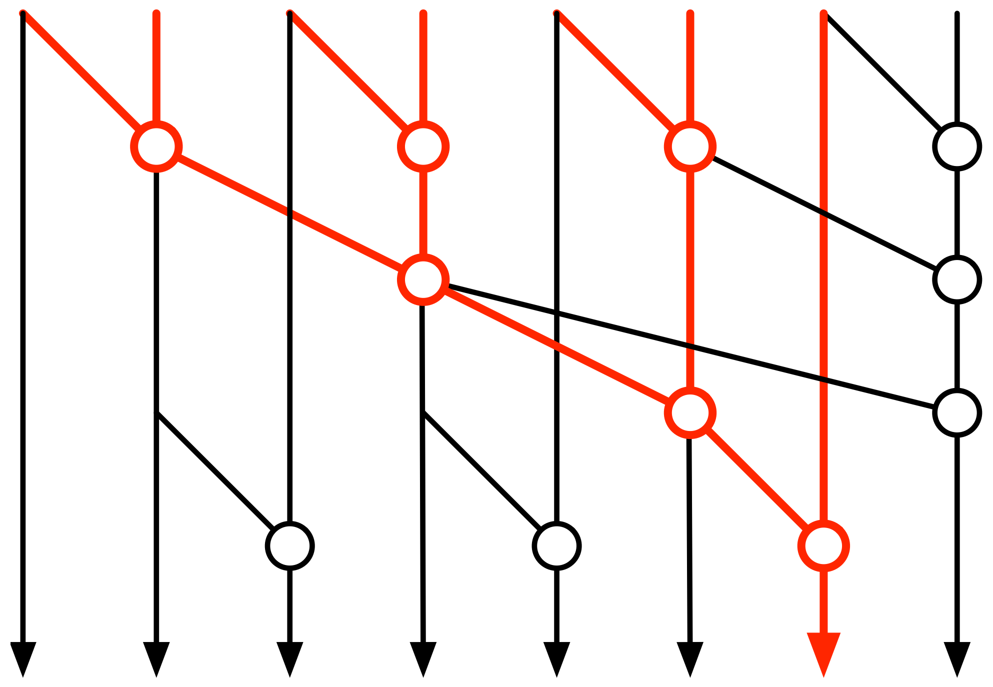


$x_1, (x_1 \oplus x_2), \cdots, (x_1 \oplus x_2 \oplus \cdots \oplus x_n)$

Top are inputs
Bottom are outputs
Circles are operation
nodes where the
operator is applied

# Scan circuits

$$x_1, x_2, \ldots, x_n$$

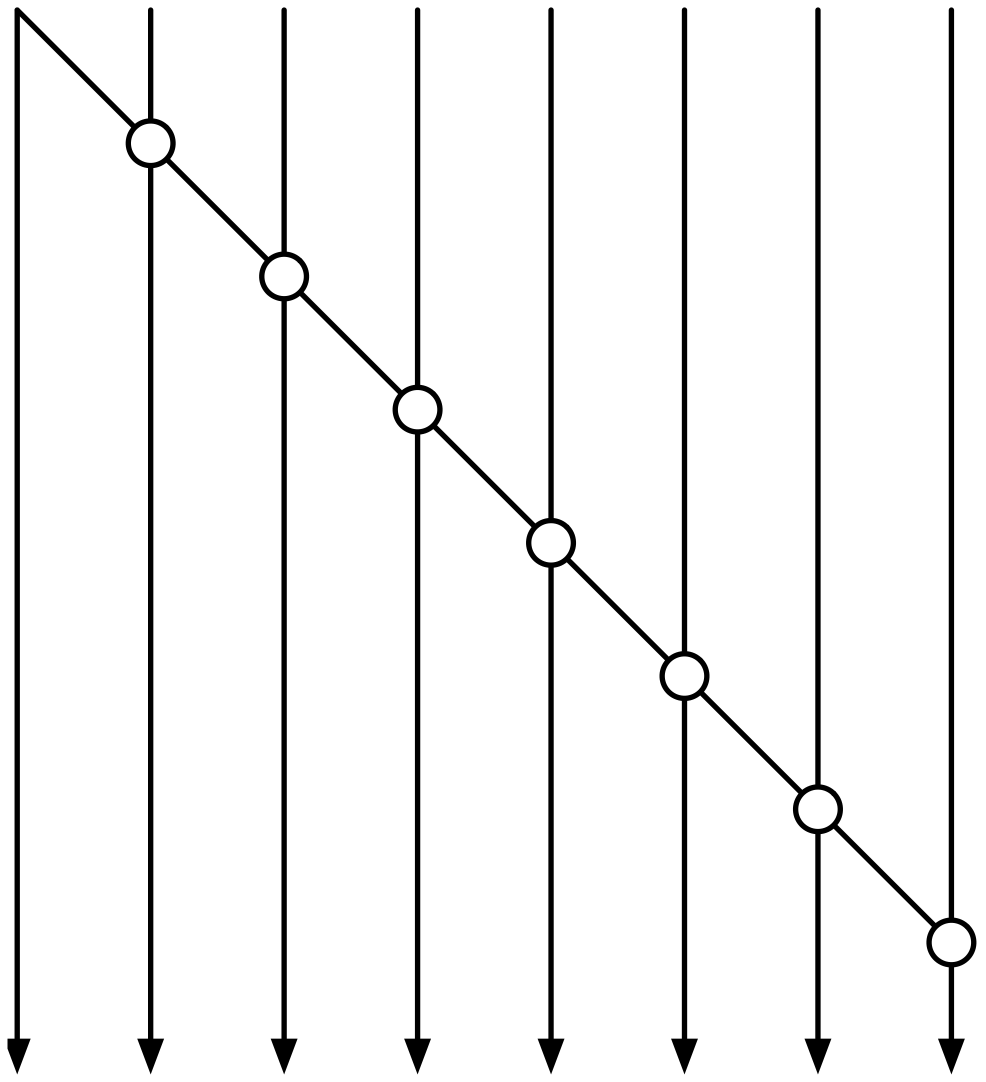


$$x_1, (x_1 \oplus x_2), \cdots, (x_1 \oplus x_2 \oplus \cdots \oplus x_n)$$

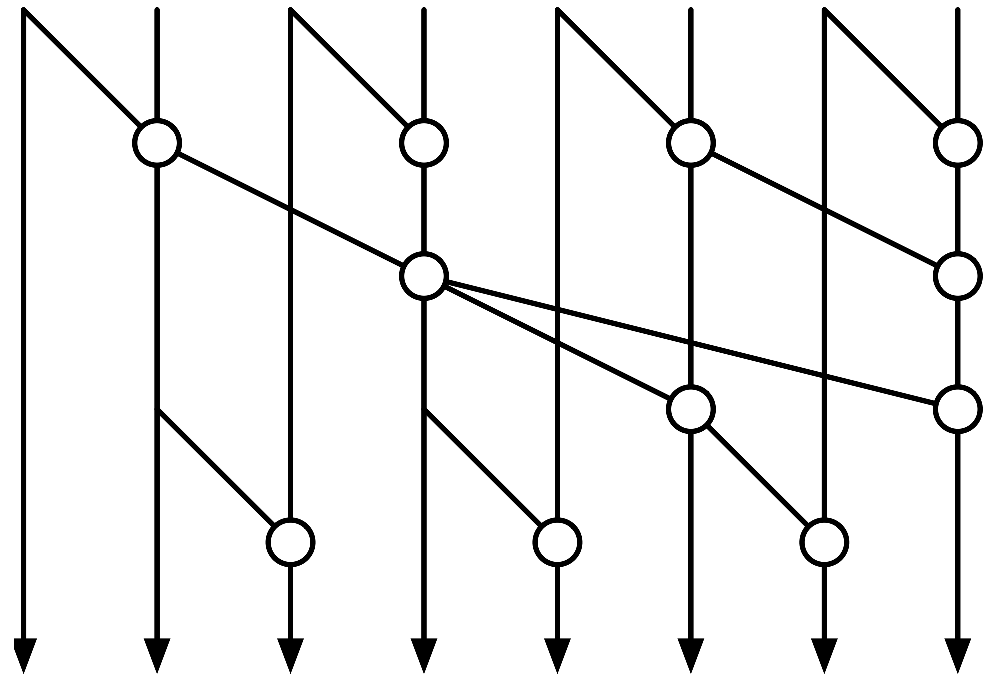
Indeed, the 7th output is the sum of the first 7 inputs.
Associativity is important here

# Scan circuits

Serial - depth O(n)



Parallel - depth O(log n)



Serial is the simplest way of calculating a scan. This is what many programming languages do by default.

Parallel is useful if your hardware supports it

# Uses of scans

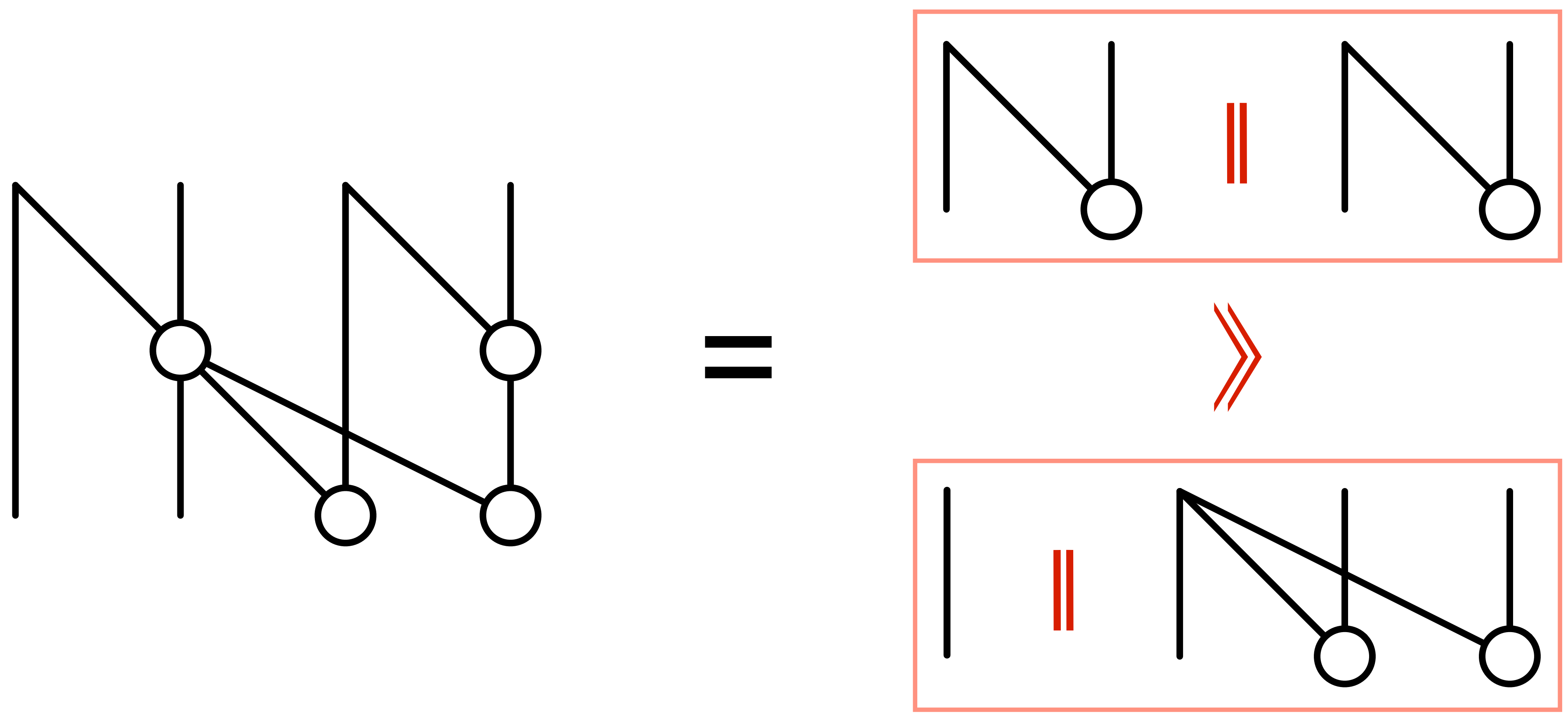
You only have to define the operator, and get the rest for free.

- Carry-lookahead adder
- Quicksort
- Calculating convex hulls
- Searching for regular expressions

## Scan algebra

- Ralf Hinze - “An algebra of scans”
- Building scan circuits from smaller components

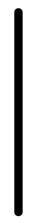
## Taking scans apart

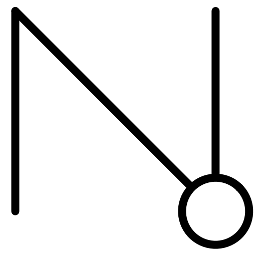


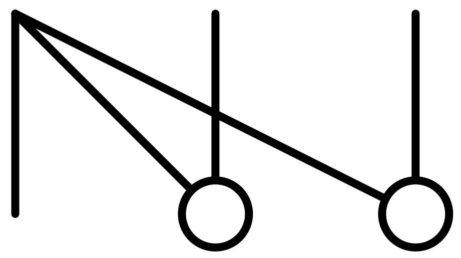
# Taking scans apart

∥ = horizontal composition

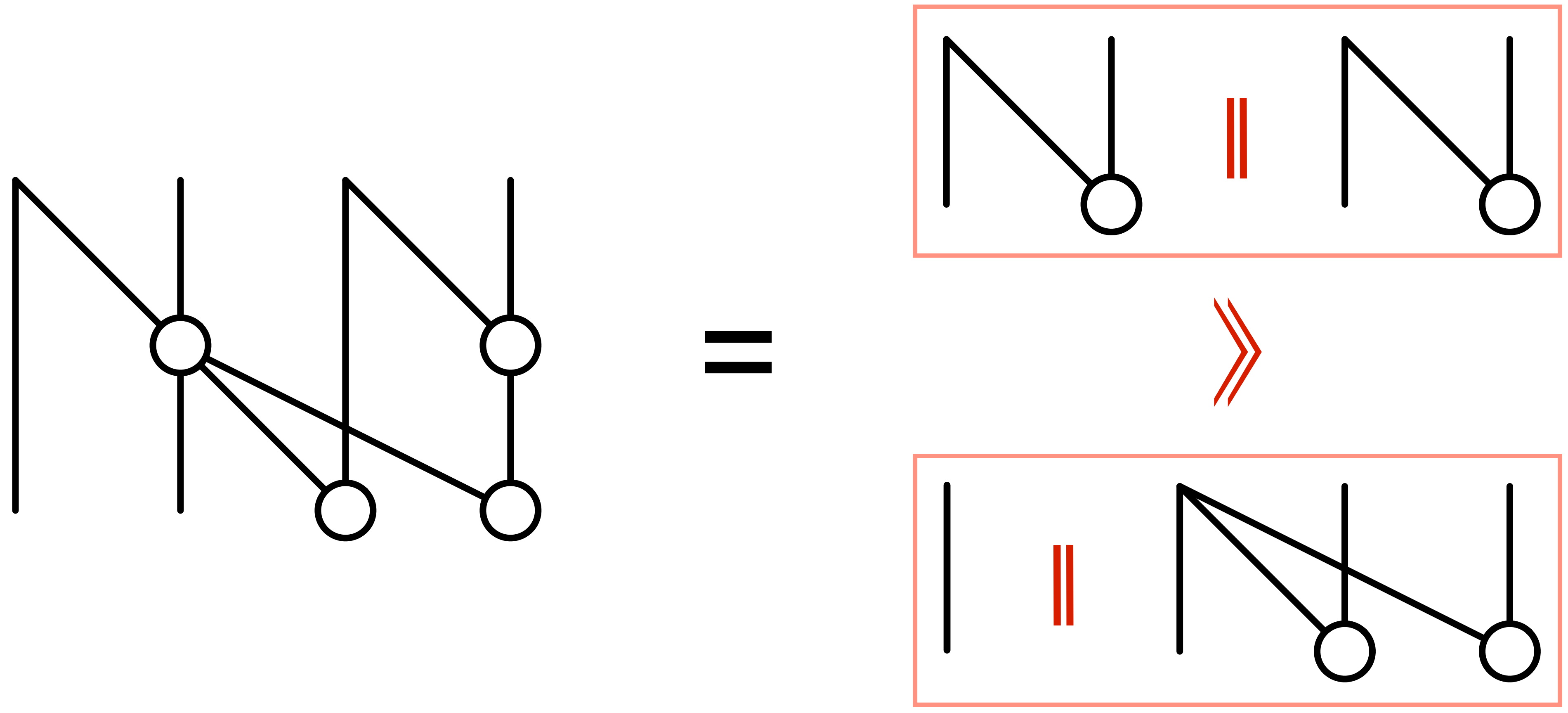
⟫ = vertical composition


= fan 1


= fan 2


= fan 3

# Taking scans apart



d-scan 4 = (fan 2 ‖ fan 2) ⟫ (fan 1 ‖ fan 3)

# Scan algebra

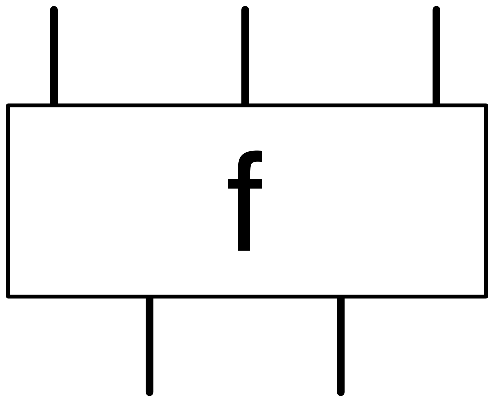
- Constructions in Ralf Hinze's scan algebra:
  - fans, ids
  - $\|$, $\gg$
  - $\multimap$, $\rightarrowtail$
- Everything else is derived from these

# PiWare

- Domain-specific language for hardware
- Embedded in Agda
- Description, simulation, and verification of circuits

# Building circuits

- A circuit in PiWare is of type **C** i o where:
  - *i* is the input size
  - *o* is the output size
- In agda syntax:
  - f : **C** 3 2



f is of type circuit-with-3-inputs-and-2-outputs

# Building circuits

f : **C** 3 2

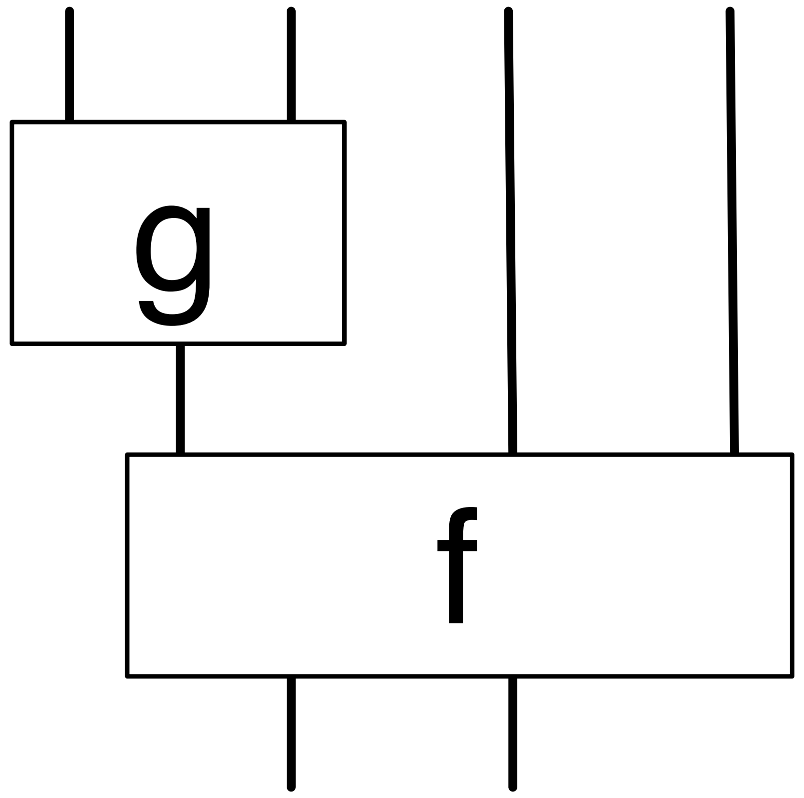
f = ?

g : **C** 2 1

g = ?

mycircuit : **C** 4 2

mycircuit = (g ∥ id 2) ⟫ f



# Constructors of $\mathbf{C}$

Plug : i ⤡ o $\rightarrow \mathbf{C}\ i\ o$

_⟫_ : $\mathbf{C}\ i\ m \rightarrow \mathbf{C}\ m\ o \rightarrow \mathbf{C}\ i\ o$

_∥_ : $\mathbf{C}\ i_1\ o_1 \rightarrow \mathbf{C}\ i_2\ o_2 \rightarrow \mathbf{C}\ (i_1 + i_2)\ (o_1 + o_2)$

Gate : (omitted)

DelayLoop : (omitted)

Plug gives a circuit where every output is connected to one of the inputs.
Use it to define id

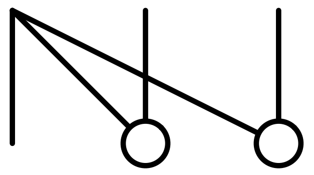
# PiWare ⇒ Scan algebra

- With PiWare, all basic constructions of the scan algebra can be implemented: fans, ids, $\|$, $\gg$, $\multimap$ and $\rightarrowtail$
- Agda can be used to verify Hinze’s proofs

# Fans in PiWare

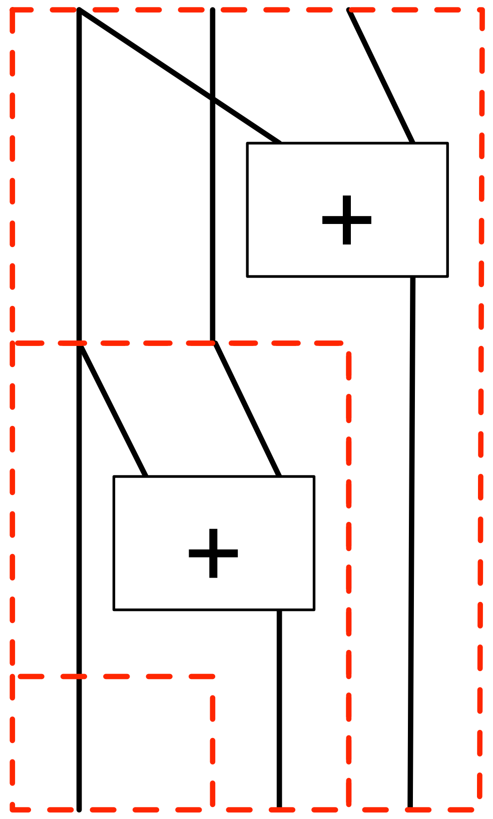
Inner rectangle is fan 1.
Middle rectangle is fan 2 (includes fan 1).
Outer rectangle is fan 3 (includes fan 2 and fan 1)

- Native in scan algebra



- In PiWare (roughly):

```
fan 0 = id 0
fan 1 = id 1
fan (2 + n) = some-plug n
  ⟫ (id (1 + n) ∥ plusC)
  ⟫ (fan (1 + n) ∥ id 1)
```



## Proofs about circuits

# Curry-Howard

- In our system, $f \approx g$ is a *type*
- The existence of a value of type $f \approx g$ means that the circuits $f$ and $g$ behave the same
- A function with return type $f \approx g$ is a *proof* that $f$ and $g$ behave the same

## Equivalence

$\mathrm{refl} : (f : \mathbf{C}\ i\ o) \to f \approx f$

$\mathrm{sym} : (f : \mathbf{C}\ i_1\ o_1) \to (g : \mathbf{C}\ i_2\ o_2) \to$
$\quad f \approx g \to g \approx f$

$\mathrm{trans} : (f : \mathbf{C}\ i_1\ o_1) \to (g : \mathbf{C}\ i_2\ o_2) \to$
$\quad (h : \mathbf{C}\ i_3\ o_3) \to$
$\quad f \approx g \to g \approx h \to f \approx h$

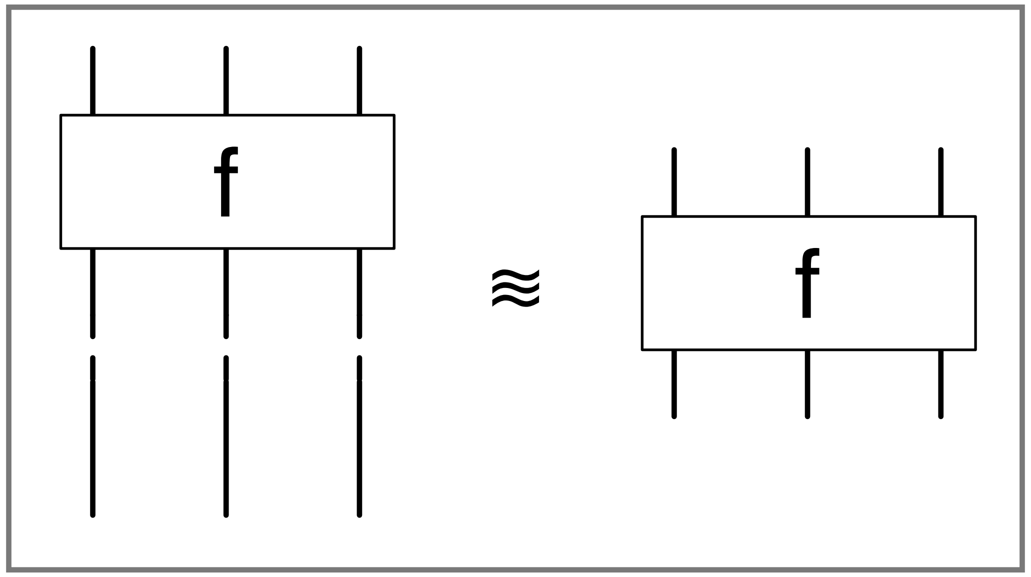
Agda syntax here.
Reflexivity takes a circuit of size i/o named f and produces a proof that this circuit is equal to itself.
Symmetry takes two circuits and a proof that the first is equal to the second. It returns a proof that the second is equal to the first.

I am omitting some non-interesting parameters, so not to scare non-agda folk

# Laws of ⟫

⟫-left-identity : (f : C i o) → (id i ⟫ f) ≈ f

⟫-right-identity : (f : C i o) → (f ⟫ id o) ≈ f



Putting an identity circuit above or below a circuit should not change its behavior.

The picture is for right-identity

# Laws of ⟫

⟫-associativity :

(f : **C** i m) → (g : **C** m n) → (h : **C** n o) →

f ⟫ (g ⟫ h) ≈ (f ⟫ g) ⟫ h

# Composability

- Parts are equal ⇒ whole is equal
- $\_⟫\text{-cong}\_ : \{f : C\ i\ m\} \to \{g : C\ m\ o\} \to \{f' : C\ i'\ m'\} \to \{g' : C\ m'\ o'\} \to f \approx f' \to g \approx g' \to f ⟫ g \approx f' ⟫ g'$
- $\text{fan-cong} : m \equiv n \to \text{fan}\ m \approx \text{fan}\ n$
- Also _∥-cong_, id-cong et cetera

If all parts of two circuits are equal, then the whole circuits are equal. Also: we can replace a part of a circuit by another part with the same behavior, the behavior of the whole stays the same.

fan-cong takes a proof that two numbers are equal

# Combining proofs

```
prf : (f : C n n) →
      (f ∥ id 0) ⟫ fan (n + 0) ≈ f ⟫ fan n
prf = (∥-right-identity f)
        ⟫-cong (fan-cong (plus-zero n))
```

Note that ⟫-cong is applied infix.

∥-right-identity f is a proof that f ∥ id 0 equals f.

## Scans

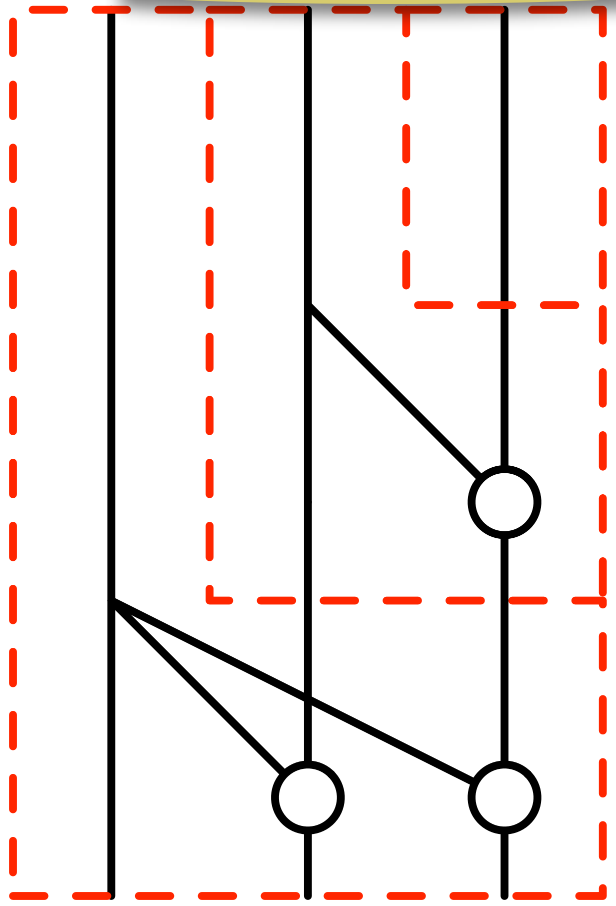
# The naive scan

Again, inner rectangle is scan 1, middle rectangle scan 2 and outer rectangle is scan 3.

Worst possible scan, maximum number of nodes and maximum depth. But the definition is straightforward and easy to work with

```
scan-suc : ∀ {n} →
  C n n → C (1 + n) (1 + n)
scan-suc {n} f = id 1 ∥ f ⟫ fan (1 + n)

scan : ∀ n → C n n
scan zero = id 0
scan (1 + n) = scan-suc (scan n)
```

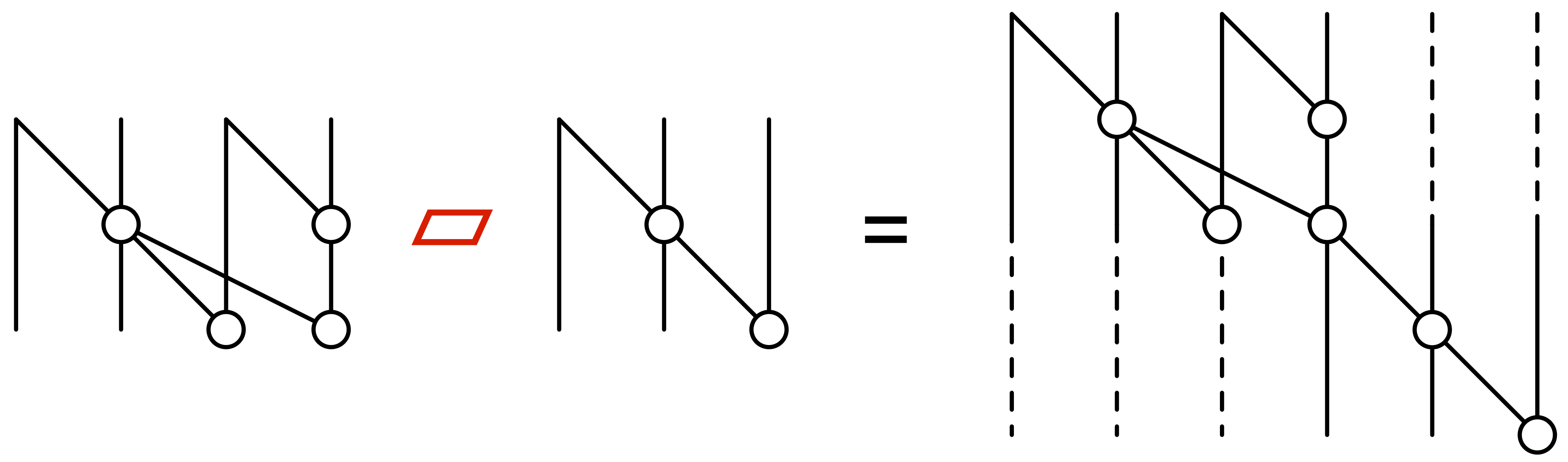


## Scans

- A circuit f is a scan if it is behaviorally equal to the naive scan
- f ≈ scan n

# Combining scans

- ▱ is diagonal composition
- It is a *scan combinator*
- If f and g are scans, then f ▱ g is also a scan

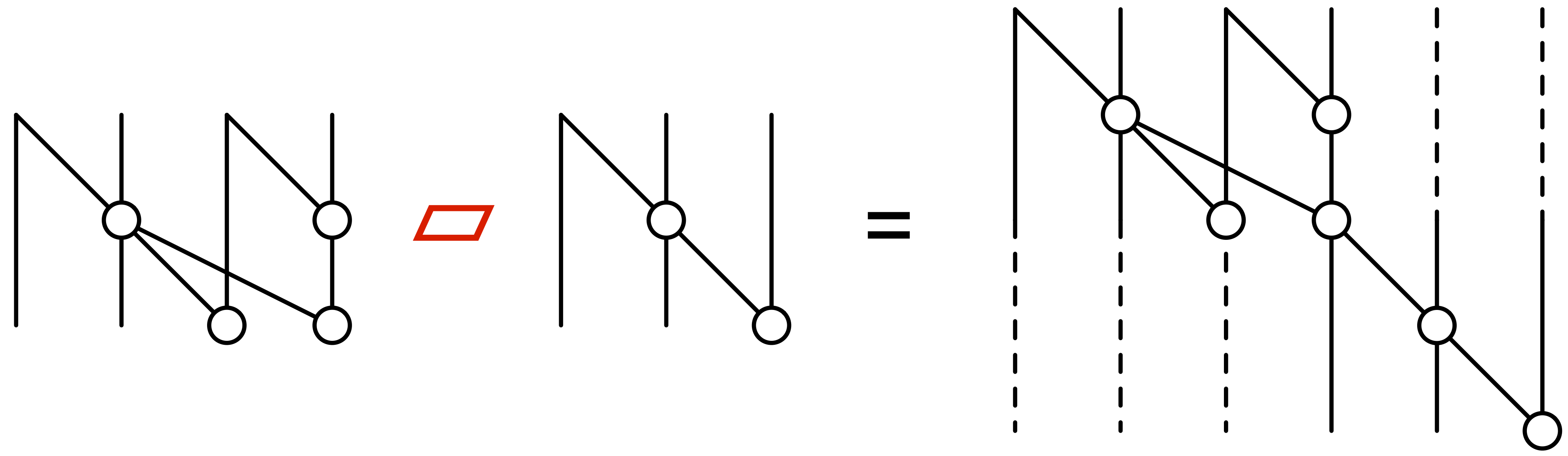


# Combining scans

A proof that combining two scans with the scan combinator indeed gives a scan. This is made easy by using the naive scan.

If you have some proof that

- ▱-combinator : ∀ m n →
  scan (suc m) ▱ scan (suc n) ≈
  scan (m + suc n)



# Scan combinators

Also, many scans can be defined by combining trivial scans (scan 0, scan 1, scan 2) with scan combinators.

- Hinze has defined more scan combinators:
  - Horizontal scan combinator ▯
  - Multi-horizontal scan combinator ⊳
- Useful for proving that big circuits are scans

# Real proofs

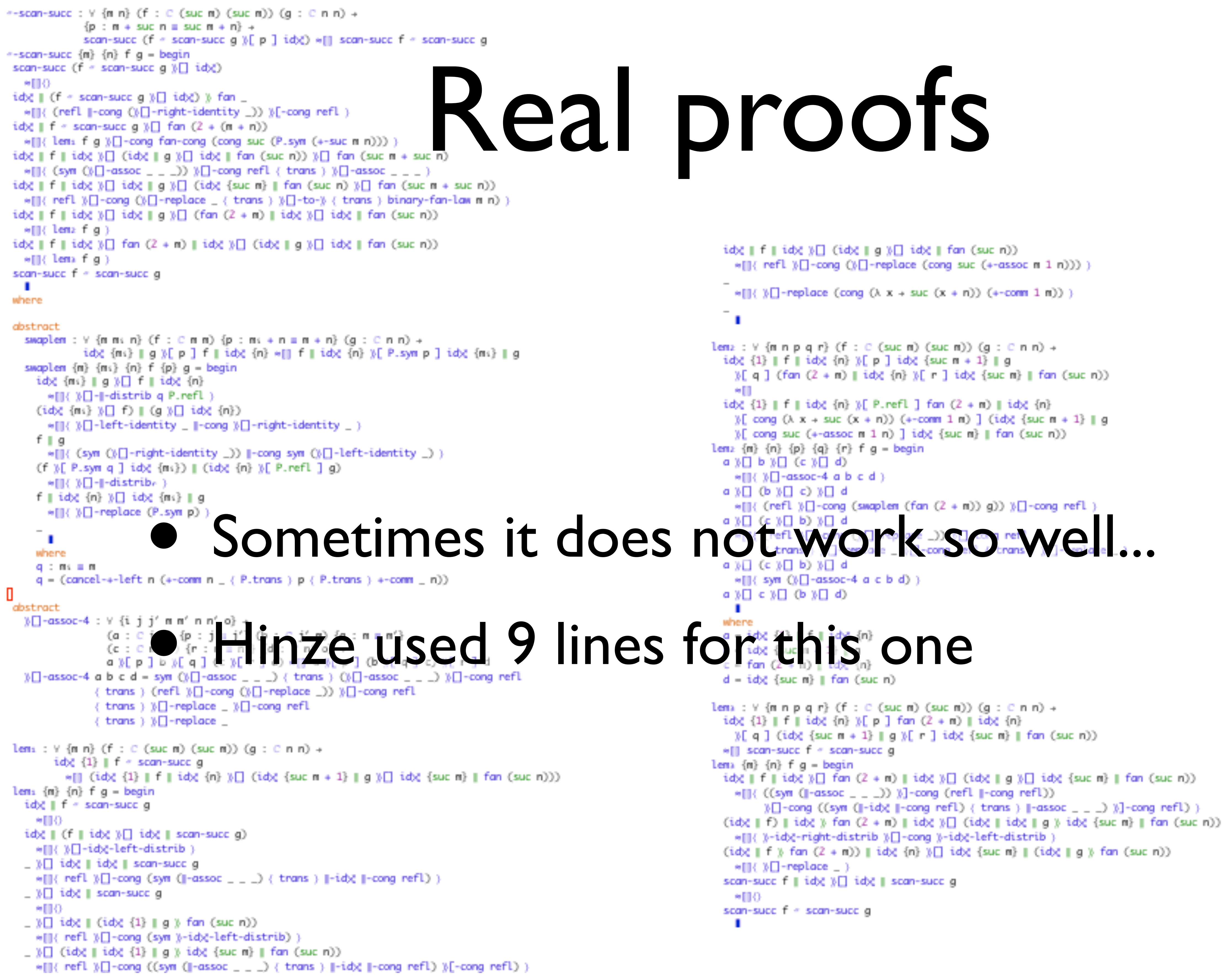
```
▯-combinator : ∀ m n →
               scan (suc m) ▯ scan n ≈ scan (suc m + n)
▯-combinator m n = begin
  scan (suc m) ▯ scan n          ≈⟨ ▯-▱ (scan (suc m)) (scan n) ⟩
  scan (suc m) ▱ scan (suc n) ≈⟨ ▱-combinator m n ⟩
  scan (m + suc n)               ≈⟨ scan-cong (+-suc m n) ⟩
  scan (suc m + n)               ∎
```

Equational reasoning here.
The left side are the terms, the first one matches the left hand side of the proof obligation and the last one matches the right hand side.
On the right side are the proofs which convert a term to the next one.

Proof that ▯ is a scan combinator
Step 1: Convert ▯ to ▱
Step 2: Use the fact that ▱ is a scan combinator
Step 3: Rewrite the size of the scan

# Real proofs

- Sometimes it does not work so well...
- Hinze used 9 lines for this one



```
⁀-scan-succ : ∀ {m n} (f : ℂ (suc m) (suc m)) (g : ℂ n n) →
              {p : m + suc n ≡ suc m + n} →
              scan-succ (f ⁀ scan-succ g ⟫[ p ] id⊗) ≈⟦⟧ scan-succ f ⁀ scan-succ g
⁀-scan-succ {m} {n} f g = begin
  scan-succ (f ⁀ scan-succ g ⟫□ id⊗)
    ≈⟦⟧⟨⟩
  id⊗ ∥ (f ⁀ scan-succ g ⟫□ id⊗) ⟫ fan _
    ≈⟦⟧⟨ (refl ∥-cong (⟫□-right-identity _)) ⟫[-cong refl ⟩
  id⊗ ∥ f ⁀ scan-succ g ⟫□ fan (2 + (m + n))
    ≈⟦⟧⟨ lem₁ f g ⟫□-cong fan-cong (cong suc (P.sym (+-suc m n))) ⟩
  id⊗ ∥ f ∥ id⊗ ⟫□ (id⊗ ∥ g ⟫□ id⊗ ∥ fan (suc n)) ⟫□ fan (suc m + suc n)
    ≈⟦⟧⟨ (sym (⟫□-assoc _ _ _)) ⟫□-cong refl ⟨ trans ⟩ ⟫□-assoc _ _ _ ⟩
  id⊗ ∥ f ∥ id⊗ ⟫□ id⊗ ∥ g ⟫□ (id⊗ {suc m} ∥ fan (suc n) ⟫□ fan (suc m + suc n))
    ≈⟦⟧⟨ refl ⟫□-cong (⟫□-replace _ ⟨ trans ⟩ ⟫□-to-⟫ ⟨ trans ⟩ binary-fan-law m n) ⟩
  id⊗ ∥ f ∥ id⊗ ⟫□ id⊗ ∥ g ⟫□ (fan (2 + m) ∥ id⊗ ⟫□ id⊗ ∥ fan (suc n))
    ≈⟦⟧⟨ lem₂ f g ⟩
  id⊗ ∥ f ∥ id⊗ ⟫□ fan (2 + m) ∥ id⊗ ⟫□ (id⊗ ∥ g ⟫□ id⊗ ∥ fan (suc n))
    ≈⟦⟧⟨ lem₃ f g ⟩
  scan-succ f ⁀ scan-succ g
    ∎
  where

  abstract
    swaplem : ∀ {m m₁ n} (f : ℂ m m) {p : m₁ + n ≡ m + n} (g : ℂ n n) →
              id⊗ {m₁} ∥ g ⟫[ p ] f ∥ id⊗ {n} ≈⟦⟧ f ∥ id⊗ {n} ⟫[ P.sym p ] id⊗ {m₁} ∥ g
    swaplem {m} {m₁} {n} f {p} g = begin
      id⊗ {m₁} ∥ g ⟫□ f ∥ id⊗ {n}
        ≈⟦⟧⟨ ⟫□-∥-distrib q P.refl ⟩
      (id⊗ {m₁} ⟫□ f) ∥ (g ⟫□ id⊗ {n})
        ≈⟦⟧⟨ ⟫□-left-identity _ ∥-cong ⟫□-right-identity _ ⟩
      f ∥ g
        ≈⟦⟧⟨ (sym (⟫□-right-identity _)) ∥-cong sym (⟫□-left-identity _) ⟩
      (f ⟫[ P.sym q ] id⊗ {m₁}) ∥ (id⊗ {n} ⟫[ P.refl ] g)
        ≈⟦⟧⟨ ⟫□-∥-distribᵣ ⟩
      f ∥ id⊗ {n} ⟫□ id⊗ {m₁} ∥ g
        ≈⟦⟧⟨ ⟫□-replace (P.sym p) ⟩
      _
      ∎
      where
      q : m₁ ≡ m
      q = (cancel-+-left n (+-comm n _ ⟨ P.trans ⟩ p ⟨ P.trans ⟩ +-comm _ n))

  abstract
    ⟫□-assoc-4 : ∀ {i j j′ m m′ n n′ o} →
                 (a : ℂ i [illegible]) {p : j [illegible] j′} [illegible] {q : m ≡ m′}
                 (c : ℂ [illegible]) {r : [illegible]} [illegible]
                 a ⟫[ p ] b ⟫[ q ] [illegible] ⟫[ p ] (b ⟫[ q ] c) ⟫[ r ] d
    ⟫□-assoc-4 a b c d = sym (⟫□-assoc _ _ _) ⟨ trans ⟩ (⟫□-assoc _ _ _) ⟫□-cong refl
                 ⟨ trans ⟩ (refl ⟫□-cong (⟫□-replace _)) ⟫□-cong refl
                 ⟨ trans ⟩ ⟫□-replace _ ⟫□-cong refl
                 ⟨ trans ⟩ ⟫□-replace _

  lem₁ : ∀ {m n} (f : ℂ (suc m) (suc m)) (g : ℂ n n) →
         id⊗ {1} ∥ f ⁀ scan-succ g
           ≈⟦⟧ (id⊗ {1} ∥ f ∥ id⊗ {n} ⟫□ (id⊗ {suc m + 1} ∥ g ⟫□ id⊗ {suc m} ∥ fan (suc n)))
  lem₁ {m} {n} f g = begin
    id⊗ ∥ f ⁀ scan-succ g
      ≈⟦⟧⟨⟩
    id⊗ ∥ (f ∥ id⊗ ⟫□ id⊗ ∥ scan-succ g)
      ≈⟦⟧⟨ ⟫□-id⊗-left-distrib ⟩
    _ ⟫□ id⊗ ∥ id⊗ ∥ scan-succ g
      ≈⟦⟧⟨ refl ⟫□-cong (sym (∥-assoc _ _ _) ⟨ trans ⟩ ∥-id⊗ ∥-cong refl) ⟩
    _ ⟫□ id⊗ ∥ scan-succ g
      ≈⟦⟧⟨⟩
    _ ⟫□ id⊗ ∥ (id⊗ {1} ∥ g ⟫ fan (suc n))
      ≈⟦⟧⟨ refl ⟫□-cong (sym ⟫-id⊗-left-distrib) ⟩
    _ ⟫□ (id⊗ ∥ id⊗ {1} ∥ g ⟫ id⊗ {suc m} ∥ fan (suc n))
      ≈⟦⟧⟨ refl ⟫□-cong ((sym (∥-assoc _ _ _) ⟨ trans ⟩ ∥-id⊗ ∥-cong refl) ⟫[-cong refl) ⟩
    id⊗ ∥ f ∥ id⊗ ⟫□ (id⊗ ∥ g ⟫□ id⊗ ∥ fan (suc n))
      ≈⟦⟧⟨ refl ⟫□-cong (⟫□-replace (cong suc (+-assoc m 1 n))) ⟩
    _
      ≈⟦⟧⟨ ⟫□-replace (cong (λ x → suc (x + n)) (+-comm 1 m)) ⟩
    _
      ∎

  lem₂ : ∀ {m n p q r} (f : ℂ (suc m) (suc m)) (g : ℂ n n) →
    id⊗ {1} ∥ f ∥ id⊗ {n} ⟫[ p ] id⊗ {suc m + 1} ∥ g
      ⟫[ q ] (fan (2 + m) ∥ id⊗ {n} ⟫[ r ] id⊗ {suc m} ∥ fan (suc n))
      ≈⟦⟧
    id⊗ {1} ∥ f ∥ id⊗ {n} ⟫[ P.refl ] fan (2 + m) ∥ id⊗ {n}
      ⟫[ cong (λ x → suc (x + n)) (+-comm 1 m) ] (id⊗ {suc m + 1} ∥ g
      ⟫[ cong suc (+-assoc m 1 n) ] id⊗ {suc m} ∥ fan (suc n))
  lem₂ {m} {n} {p} {q} {r} f g = begin
    a ⟫□ b ⟫□ (c ⟫□ d)
      ≈⟦⟧⟨ ⟫□-assoc-4 a b c d ⟩
    a ⟫□ (b ⟫□ c) ⟫□ d
      ≈⟦⟧⟨ (refl ⟫□-cong (swaplem (fan (2 + m)) g)) ⟫□-cong refl ⟩
    a ⟫□ (c ⟫□ b) ⟫□ d
      ≈⟦⟧⟨ (refl ⟫□-cong [illegible] ⟫□-replace _)) ⟫□-cong refl
           ⟨ trans ⟩ [illegible] ⟫□-replace _ ⟫□-cong refl ⟨ trans ⟩ ⟫□-replace _ ⟩
    a ⟫□ (c ⟫□ b) ⟫□ d
      ≈⟦⟧⟨ sym (⟫□-assoc-4 a c b d) ⟩
    a ⟫□ c ⟫□ (b ⟫□ d)
      ∎
    where
    a = id⊗ {1} ∥ f ∥ id⊗ {n}
    b = id⊗ {suc m + 1} ∥ g
    c = fan (2 + m) ∥ id⊗ {n}
    d = id⊗ {suc m} ∥ fan (suc n)

  lem₃ : ∀ {m n p q r} (f : ℂ (suc m) (suc m)) (g : ℂ n n) →
    id⊗ {1} ∥ f ∥ id⊗ {n} ⟫[ p ] fan (2 + m) ∥ id⊗ {n}
      ⟫[ q ] (id⊗ {suc m + 1} ∥ g ⟫[ r ] id⊗ {suc m} ∥ fan (suc n))
    ≈⟦⟧ scan-succ f ⁀ scan-succ g
  lem₃ {m} {n} f g = begin
    id⊗ ∥ f ∥ id⊗ ⟫□ fan (2 + m) ∥ id⊗ ⟫□ (id⊗ ∥ g ⟫□ id⊗ {suc m} ∥ fan (suc n))
      ≈⟦⟧⟨ ((sym (∥-assoc _ _ _)) ⟫]-cong (refl ∥-cong refl))
           ⟫□-cong ((sym (∥-id⊗ ∥-cong refl) ⟨ trans ⟩ ∥-assoc _ _ _) ⟫]-cong refl) ⟩
    (id⊗ ∥ f) ∥ id⊗ ⟫ fan (2 + m) ∥ id⊗ ⟫□ (id⊗ ∥ id⊗ ∥ g ⟫ id⊗ {suc m} ∥ fan (suc n))
      ≈⟦⟧⟨ ⟫-id⊗-right-distrib ⟫□-cong ⟫-id⊗-left-distrib ⟩
    (id⊗ ∥ f ⟫ fan (2 + m)) ∥ id⊗ {n} ⟫□ id⊗ {suc m} ∥ (id⊗ ∥ g ⟫ fan (suc n))
      ≈⟦⟧⟨ ⟫□-replace _ ⟩
    scan-succ f ∥ id⊗ ⟫□ id⊗ ∥ scan-succ g
      ≈⟦⟧⟨⟩
    scan-succ f ⁀ scan-succ g
      ∎
```

# Results

- Scans à la Hinze can be formalized in PiWare
- Useful additions to PiWare
- Our proofs usually follow the same structure as Hinze's proofs

Even the very long one on the previous page follows the same structure as Hinze in his paper.

# Work i did not do

- There are still some holes in the proofs
- Proofs about depth-optimality
- Proofs about size-optimality

These proofs are in Hinze's paper, but i did not get to that point.

# Thanks

- Scan algebra - Fans, scans, ∥, ⟫
- PiWare - Plug, ∥, ⟫
- Proofs - ≋, composability
- Scans - The naive scan, scan combinators